Reducing Write Barrier Overheads for Orthogonal Persistence

Yilin Zhang*
University of Tokyo
Tokyo, Japan
elliot@csg.ci.i.u-tokyo.ac.jp

Omkar Dilip Dhawal*
IIT Madras
Chennai, India
omkar@cse.iitm.ac.in

V. Krishna Nandivada
IIT Madras
Chennai, India
nvk@iitm.ac.in

Shigeru Chiba
University of Tokyo
Tokyo, Japan
chiba@acm.org

Tomoharu Ugawa
University of Tokyo
Tokyo, Japan
tugawa@acm.org

Abstract

Orthogonal persistence implemented with non-volatile memory (NVM) allows the programmers to easily create persistent containers, which are container data-structures preserved even after the process terminations due to a system crash. However, the state-of-the-art technique of its implementation in multithreaded languages rely on the MFENCE instruction, which limits out-of-order execution. This overhead is applied regardless of the use of persistent objects. We propose a technique that does not disturb out-of-order execution. Instead, we let the thread that is attempting to make an object persistent synchronize with all the other threads by handshaking. Furthermore, we propose a technique to eliminate the redundancy of that synchronization by a novel static analysis called persistence-aware escape analysis. We implemented both the proposed techniques in RBP (replication based persistency) implemented in the HotSpot VM of OpenJDK. As a result of our evaluation, we observed that the execution speed was faster than RBP by 23.0% when a program did not use persistent objects, and it was only 10.6% slower than the standard HotSpot VM, which does not support orthogonal persistence. When a program used persistent objects, execution speed was almost the same as RBP. These results demonstrate that orthogonal persistence using NVM can be implemented in a practical way.

CCS Concepts: • **Hardware → Non-volatile memory**; • **Software and its engineering → Just-in-time compilers**.

*Equal contributions by both.

SLE '24, October 20–21, 2024, Pasadena, CA, USA

ACM ISBN 979-8-4007-1180-0/24/10
https://doi.org/10.1145/3687997.3695646

Keywords: Non-Volatile Memory, Concurrency, Memory Management, Orthogonal Persistence, Escape Analysis

ACM Reference Format:
Yilin Zhang, Omkar Dilip Dhawal, V. Krishna Nandivada, Shigeru Chiba, and Tomoharu Ugawa. 2024. Reducing Write Barrier Overheads for Orthogonal Persistence. In *Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering (SLE '24), October 20–21, 2024, Pasadena, CA, USA*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3687997.3695646

1 Introduction

The objects in byte-addressable non-volatile memory (NVM) can be accessed by the program in the same way as those in DRAM, and they themselves are persistent after their data are written back to NVM from the CPU cache. However, NVM is not as fast as DRAM, though it is faster than SSDs. Therefore, hybrid systems of DRAM and NVM are used in practice where *persistent* objects reside in NVM while *volatile* objects reside in DRAM.

Such hybrid systems bring an interesting challenge to the software side to maintain the integrity and consistency of the data in NVM so that whenever the system crashes during an execution, we can recover (and resume) the execution using the data in NVM. For example, programs must not create pointers from objects in NVM to objects in DRAM, which become dangling pointers after a crash. Therefore, programs must place the objects that such pointers point to in NVM. In addition, the programs must control the order of writes to NVM, which involves writing back cache lines accordingly. Manually controlling when and which objects must be made persistent is a tedious and error-prone task for the programmer. To address this problem, various programming models [25, 30] have been proposed.

Orthogonal persistence model [3, 30] is one of the most programmer-friendly programming models. In this model, the persistence of objects is decided by reachability; objects reachable from the user-defined *persistent roots* are persistent. The locations of the data and the order of writes are managed by the system accordingly based on the persistence of objects.

In essence, the programmer only defines the persistent roots to use persistent objects.

AutoPersist [31] and replication-based persistence (RBP) [25] are the state-of-the-art of orthogonal persistence using NVM in managed runtimes. They copy objects in DRAM to NVM when they become reachable from the persistent roots. Unfortunately, both systems incur a large overhead at every write to objects. As we will show in Section 5, programs suffer from 43.7% overhead on average for RBP even when they do not make any objects persistent. This is because they rely on serialized memory accesses by the MFENCE instruction executed in runtime checks that are carried out in write barriers. These checks are to detect the case where the object to be written is being copied to NVM by another thread in multithreaded environments under a weak memory consistency model, such as x86 total store ordering (TSO) [16].

In this paper, we reduce the write barrier overhead by combining two novel techniques. First, we remove MFENCE from the runtime check. Instead, when a thread copies an object, the thread waits till all other threads acknowledge this attempt using handshakes. Though threads have to answer to handshakes, its overhead is negligible [1].

Second, we use a novel extension to the classical escape analysis (called persistence-aware escape analysis) to reduce these persistence-related overheads. We identify the field-write instructions where no thread will write to the objects that are made persistent. Further, to avoid the overhead of expensive analysis at runtime, we take inspiration from prior work [14, 37] and perform our proposed analysis during static compilation time, and then use the analysis results in the JVM.

We implemented the static components of our proposed scheme in the Soot compiler framework [40], and the runtime components of our scheme in RBP (replication based persistence) implemented in the OpenJDK HotSpot VM. We evaluated these implementations to verify that the proposed write barrier reduces overhead for programs that do not use persistent objects very often. As a result, we observed that the execution speed was faster than RBP by 23.0% when a program did not use persistent objects, and it was only 10.6% slower than the standard HotSpot VM, which does not support orthogonal persistence. We also evaluated the penalty of persistence-related overheads introduced by our write barrier and the impact of the proposed static analysis. We show that programs that do use persistent objects incurred 37.9% of persistence-related overheads on average. However, 52.0% of them were eliminated by our static analysis. These results demonstrate that orthogonal persistence using NVM can be implemented practically and efficiently.

The rest of the paper is organized as follows. Section 2 describes background and the the problem of orthogonal persistence using NVM. In Sections 3 and 4 we present the proposed write barriers and static analysis, respectively. We evaluate our proposal in Section 5, and discuss related work in Section 6. Finally, we conclude in Section 7.

[1] Lin et al. [22] reported this overhead to be 1.9%.

```
1  class Program {
2    @durable_root static Pair assocList;
3    @durable_root static int count;
4    static void assoc(Object k, Object v) {
5      Pair p = new Pair(); // p is volatile object
6      p.head = k;
7      p.tail = v;
8      Pair r = new Pair(); // r is volatile object
9      r.head = p;
10     r.tail = Program.assocList;
11     Program.assocList = r; // p and r become persistent
12     Program.count = Program.count + 1;  } }
```

Figure 1. Program constructing a persistent association list.

2 Background

2.1 Orthogonal Persistence

In the orthogonal persistence model [3, 30] the programmers specify the roots of persistent data structures. They are called *persistent roots*. Objects reachable from the persistent roots are guaranteed to be persistent by the system. In such systems, objects are created as non-persistent, which we say *volatile*. They dynamically become persistent when they become reachable from the persistent roots.

Figure 1 shows a typical example of a Java program that constructs a persistent association list comprising persistent objects. It shows a common case where locally constructed objects are stored in persistent containers. In the example, the static fields `assocList` and `count` are the persistent roots. The field `assocList` points-to the head of the association list. It is a list of association cells, each of which contains a key-value pair in its `head` and `tail` fields. The field `count` holds the number of elements in the association list.

The `assoc` method creates a new association cell containing a key-value pair (`k`, `v`) and appends it to the list. `Pairs` pointed-to by `p` and `r` are volatile when they are created at lines 5 and 8. Therefore, assignments to their fields do not make any objects persistent. Note that volatile objects can have references to persistent objects. Thus, the assignment at line 10 does not make the object pointed-to by `r` persistent, though `assocList` points to a persistent object.

At line 11, a volatile object (pointed-to by `r`) is assigned to the persistent root `assocList`. Therefore, the object pointed-to by `r` is made persistent before the assignment. The object pointed-to by `p` is also made persistent because it is reachable from `r`. At line 12, another persistent root `count` is updated. In these systems, writes to persistent roots and persistent objects are reflected in NVM in an order that does not contradict the happens-before order [23], which includes the program order. Therefore, whenever the system crashes, it is guaranteed that at least the first `count` elements in the association list are valid.

Another typical use case is minimally ordered durable data structures [15], which are variants of immutable data structures, such as hash-array mapped tries [4], compressed hash-array mapped prefix-trees [34], and relaxed radix balanced trees [36]. Immutable data structures cannot be updated in place. Rather, a new version is created by copying unchanged parts from the existing version to be updated while sharing totally unchanged nodes among the old and new versions. The newly created nodes are initialized while they are volatile. They are made persistent when the root of the immutable data structure is assigned to the persistent root.

From our experience, we have observed that (i) writing to persistent objects, which may make some objects persistent, is not frequent, and (ii) objects tend to be initialized while they are thread-local.

2.1.1 Efficiency Issues with Orthogonal Persistence.

The hybrid memory systems discussed earlier in this section create objects in DRAM and copy them to NVM to make them persistent. We call the thread that copies the object a *copier thread*. The copier thread copies without stopping the world. Therefore, it is possible that another thread may write to the object being copied. We call this writing thread a *writer thread*.

To resolve the race, both PBR and AutoPersist use an MFENCE instruction on the execution path that is (almost) always executed, or *hot path*, of the write barrier. This applies a large overhead to programs regardless of whether the programs create persistent objects.

In the rest of this subsection, we explain why MFENCE is necessary. RBP relies on the writers to write the same value to both DRAM and NVM. However, this double-write is conditional; the writer writes only when the object that it is writing to has a replica. The replica is installed by the copier concurrently with the writer. MFENCE is necessary for the writers to acknowledge that the copier has installed the replica.

AutoPersist [31] makes the copier fail when this race is detected. The copier sets a *copying* flag on the object to be copied. The writer thread resets it before it writes to the object. After copying, if the copier finds the flag reset, the copier retries. The copying bit must be accessed using an expensive atomic instruction. Although an optimization to eliminate atomic instructions in most cases is also proposed, the optimization needs to read the object header *after* writing to a volatile object (see Section 6.3 in Shull et al. [31], for details). In a weak memory consistency model, such as x86 TSO, the load instruction of the object header may be reordered with the preceding store. To ensure memory order, an MFENCE instruction is required every time it writes to a volatile object.

```
1 putfield(o, f, v): // performs o.f = v
2   o[f] = v
3   MFENCE
4   o′ = o.replica
5   if o′ == NULL: return
6   make_persistent(v)
7   v′ = v.replica
8   o′[f] = v′
9   CLWB(&o′[f])
```

Figure 2. Write barrier of RBP.

2.2 Replication Based Persistence

2.2.1 Overview.

RBP makes objects persistent by creating their *replicas* in NVM while continuing to use objects in DRAM. We call an object in DRAM a *DRAM copy* of the object. A DRAM copy and its replica are loosely synchronized; except during being updated, their primitive type fields have the same values and reference type fields have references to the DRAM copies and replicas of the same objects.

To keep them synchronized, RBP uses write barriers that write to both the DRAM copy and the replica for persistent objects. Furthermore, RBP does not use read barriers. Reading from either object yields the same value or a reference to the same object under the assumption that the program is data-race-free. RBP does not support programs with data race on normal fields; racy access is allowed only to volatile fields and through the `java.util.concurrent.atomic` package. Because reading from NVM is slower than reading from DRAM, the program reads from DRAM copies. To do that, RBP maintains that the thread stacks and registers always have references to DRAM copies.

2.2.2 Write Barrier.

Figure 2 shows the write barrier of the RBP. The `putfield` function writes `v` to the field `f` of the object `o`. `putfield` assumes that `v` is a non-null reference. The thread is a writer while it is executing `putfield`. The writer writes to the DRAM copy at line 2, and writes to the replica at line 8. In this paper, we consistently use the name x' for the replica of x.

Before writing to the replica, `putfield` checks if the object has a replica or not at line 5. This check depends on the value read on line 4. To prevent this read from being reordered with the write at line 2, the MFENCE instruction is necessary.

Because `v` may be volatile, the writer ensures that `v` is persistent so that a persistent object cannot have references to volatile objects. The `make_persistent` function called at line 6 makes `v` persistent. The thread is called a copier when it is executing `make_persistent`. After writing to the replica, which is in NVM, the writer writes the cache line back to NVM using the CLWB instruction[2].

2.2.3 Traversing Transitive Closure.

Figure 3 shows the pseudo code to make objects persistent. `make_persistent`

[2]We do not present SFENCE instructions to complete CLWB in this paper because it is not relevant to our proposal.

```
1  make_persistent(r):
2    local worklist
3    if shade(r) == TRUE:
4      worklist.add(r)
5      while not worklist.empty():
6        o = worklist.remove()
7        copy_to_replica(o)
8    SYNC()
9
10 shade(o):
11   o' = alloc_NVM(sizeof(o))
12   set_responsible_thread(o', current_thread)
13   if CAS(&o.replica, NULL, o') == SUCCESS: return TRUE
14   o' = o.replica
15   t = responsible_thread(o')
16   // Thread t is making o persistent
17   if t != current_thread: depends_on(t)
18   return FALSE
```

Figure 3. `make_persistent` of RBP.

traverses the object graph from the given argument `r` and makes all reachable objects persistent. We call this `r` a *closure root*. Like tracing GC, it traverses using a mark stack `worklist`. The objects that have replicas are considered visited. The function `shade` called at Line 3 creates a replica for the given object, `r`. If it successfully installs a pointer to the replica, it returns `TRUE`. Otherwise, it means that the object already has a replica.

After it successfully installs the replica pointer, the function `copy_to_replica` called on line 7 fills the contents of the replica by copying from the DRAM copy. The function `copy_to_replica` implements the Transactional Sapphire's concurrent copying protocol [39] to resolve the race between the copier making an object persistent and the writer writing to the object. This protocol guarantees that any write made by the writer is eventually propagated to NVM. That is, the writer writes the same value to NVM (line 8 in Figure 2), or the copier copies the written value to NVM (in `copy_to_replica`). For ensuring correctness, `copy_to_replica` includes a verify step which ensures that the object being copied was not modified by another thread. The function `copy_to_replica` also calls the function `shade` on the objects that are directly pointed-to by the fields of the object being copied, and adds them to the `worklist`.

This protocol relies on the writers work, which is enabled after the writer recognizes that the replica pointer is installed. That is why it uses MFENCE instruction in the write barrier.

In a multi-threaded program, multiple copiers may call `make_persistent` on different closure roots simultaneously. It is possible that the transitive closures of these closure roots have an intersection. In such cases, two copiers create a thread group and perform a barrier synchronization by calling `SYNC` at the end of `make_persistent` so that neither thread completes the write barrier before the reachable object becomes persistent.

`Vars`	= Set of variables.
`SplObj`	= $\{E_{obj}\}$
`AObjs`	= Set of abstract objects $\cup$ `SplObj`
Escape	= Set of explicitly thread-escaping objects
ρ	: `Vars` $\rightarrow P($`AObjs`$)$
σ	: `AObjs` $\times$ `Fields` $\rightarrow P($`AObjs`$)$

Figure 4. Sets and maps used to maintain points-to + escape information at each program location. We use $P(X)$ to denote the power set of X.

In `shade`, the copier thread installs the replica using an atomic compare-and-swap (CAS) instruction because a different copier may attempt to install a replica on the same object. The copier that successfully installs the replica is called the *responsible thread* of the object and is responsible for copying the object. The thread ID of the responsible thread is recorded in the replica on line 12. This thread ID is used to detect the intersection of transitive closure on line 15.

2.3 Escape Analysis

Escape analysis [9, 42] is a very popular data-flow analysis. With the help of escape analysis, it is possible to identify objects that are guaranteed to be thread local. We now give a brief idea about the standard escape analysis relevant to this paper.

In this manuscript, we identify each abstract object allocated at Line x, by the symbol O_x (for example, the variable `p` in Figure 1 points to O_5). Thus, O_x represents all the objects that may be allocated at that line, during program execution. In addition to these abstract objects, we use an additional special abstract object E_{obj} that summarizes all the objects that are explicitly passed to other threads.

An interesting property of the special objects is that dereferencing any field of such object returns the same object. We will look at flow-sensitive analysis to handle cases where an object may be thread-local for a part of the program. In the analysis, at each program point, we maintain points-to information using the maps abstract-stack (ρ) and abstract-heap (σ) as shown in Figure 4. The *Escape* set contains the objects that may escape to other threads via the arguments passed to the thread constructor.

Figure 5 summarises the standard set of flow-functions used for the combined points-to and escape analysis. The $\rho_{out}(x)$ is set to the $\rho_{in}(y)$, after processing a copy statement. When a field of an object is dereferenced (in a statement like $x = y.f$), $\rho_{out}(x)$ will include all the objects that may be reachable from $y.f$. If y may point to the special object E_{obj} then $\rho(x)$ will also point to those special object(s).

In the case of field writes (of the form $x.f = y$), we add the objects pointed by y to the points-to set of $x.f$ in the σ map. If the object pointed to by x may-escape, then this information is propagated to all the objects in the points-to-closure of y. The macro 'propagate' (Var v, Set S, Obj o) can

Statement	Rule
(copy) $x = y$	$\rho_{out}(x) = \rho_{in}(y)$
(load) $x = y.f$	$\forall o \in \rho_{in}(y)$, if $o \in$ `SplObj` then $\rho_{out}(x) \bigcup= \{o\}$ else $\rho_{out}(x) \bigcup= \sigma_{in}(o, f)$
(store) $x.f = y$	1.$\forall o \in (\rho_{in}(x) - $ `SplObj`$)$, $\sigma_{out}(o, f) \bigcup= \rho_{in}(y)$ 2.if $\rho_{in}(x) \cap Escape \neq \phi$ then propagate(y, $Escape$, E_{obj})
(static load) $x = A.g$	$\rho_{out}(x) = \{E_{obj}\}$
(static store) $A.g = x$	propagate(x, $Escape$, E_{obj})

(alloc) $x =$ `new A(args)`	if (`A` is a thread class)
	$\rho_{out}(x) = \{E_{obj}\}$
	propagate(arg, $Escape$, E_{obj})
	// invoked for each arg
	else $\rho_{out}(x) = \{O_l\}$
	// say the line number is l

Figure 5. Flow functions for different statements to perform combined points-to and escape analysis. Assume that ρ_{in}, and σ_{in} are the ρ and σ maps, respectively, before processing the statement. Similarly, ρ_{out}, and σ_{out} are the ρ and σ maps, respectively, after processing the statement.

be defined by the two-step procedure defined below:
(a) $S \bigcup= ptsToClosure(\rho_{in}(v))$
(b) $\forall o \in ptsToClosure(\rho_{in}(v)), \forall f \in Fields, \sigma(o, f) \bigcup= \{o\}$
The method $ptsToClosure(X)$ computes the union of the points-to-closures for each object present in the set X.

Handling of static fields: Reading a static field always returns E_{obj}. Writing to a static field, leads to the propagation of escape information on the points-to-closure of the rhs.

An object allocation (at line number l) can lead to the creation of a regular object ($\rho_{out}(x)$ includes O_l), or a thread object (the escape information is propagated to the points-to-closure of the arguments).

We briefly illustrate the rules using the example in Figure 1. Assume that before reaching line 5, $\rho = \{\mathsf{k} \rightarrow \{O_k\},\ \mathsf{v} \rightarrow \{O_v\}\}$, and both the *Escape* set and σ map is empty. After Line 5, only the ρ map is updated to $\{\mathsf{k} \rightarrow \{O_k\},\ \mathsf{v} \rightarrow \{O_v\},\ p \rightarrow \{O_5\}\}$. Line 6 and Line 7 are store statements and hence only σ map is updated to $\{O_5.head \rightarrow \{O_k\},\ O_5.tail \rightarrow \{O_v\}\}$ after Line 7. After processing the static-load at Line 10, we update the σ map to $\{O_5.head \rightarrow \{O_k\},\ O_5.tail \rightarrow \{O_v\},\ O_8.head \rightarrow \{O_5\},\ O_8.tail \rightarrow \{E_{obj}\}\}$ after Line 10. Finally after the static-store at Line 11, the σ map is updated to $\{O_5.head \rightarrow \{O_k, E_{obj}\},\ O_5.tail \rightarrow \{O_v, E_{obj}\},\ O_8.head \rightarrow \{O_5, E_{obj}\},\ O_8.tail \rightarrow \{E_{obj}\}\}$ and Escape to $\{O_5, O_8, O_k, O_v\}$.

3 Copier-Wait Write Barrier

In the original RBP, the writer thread waits for its write to the DRAM copy to become visible. We call this approach the *writer-wait* approach. In this section, we propose the *copier-wait* approach, where we shift the overhead from the writers to the copiers.

3.1 Overview

In the copier-wait approach, we remove MFENCE at line 3 from the write barrier in Figure 2. Instead, the copier marks objects to be copied and waits for the other threads to *recognize* the mark. Here, we say that a thread recognizes a mark when the mark becomes visible to the thread and all the in-flight stores of the thread become globally visible.

Except for the absence of MFENCE, the writer's code of the writer barrier is the same as the writer-wait approach. In contrast, the copier's code, `make_persistent`, is different. Figure 7 shows its pseudo code. `shade` and `copy_to_replica` are the same as the writer-wait approach, shown in Figure 3.

In the copier-wait approach, `make_persistent` consists of the mark and copy phases. The mark phase (from line 7) determines the transitive closure of the given closure root, `r`. The objects in the transitive closure are marked, and the copier waits for the other threads to recognize the marks. Because the object graph may be mutated by other threads while the copier is waiting, the copier keeps tracks of the marked objects by using an array called *gray array*.

In the copy phase (from lien 12), it copies the contents of the objects marked in the mark phase one by one. Each object is copied in the same way as the writer-wait approach using `copy_to_replica` shown in Figure 3.

3.2 Write Barrier

Instead of introducing a separate mark bit, we use the presence of the replica to indicate that the object is being copied. If the replica pointer field is not null, the object is regarded as marked.

In the writer-wait approach, the presence of the replica pointer means that the copier may be copying the contents because the copier starts copying as soon as it installs the replica pointer. Therefore, MFENCE is necessary for the writer to see the latest value in the replica pointer field in a weak memory consistency model.

In the copier-wait approach, the presence of the replica does not mean that the copier is copying. The copier starts to copy after the replica pointer is recognized by the writer. Thus, if the writer reads an old value of the replica pointer field, it can behave on the basis of the old value because the copier is waiting. Therefore, MFENCE is not necessary on the hot path.

If the writer observes a replica pointer on line 4 in Figure 2, the copier may be copying. In this case, the writer writes to the replica at line 8 in Figure 2 following the concurrent copying protocol.

3.3 Ragged Synchronization

The copier performs a handshake with all the other threads. This ensures that the installed replicas are recognized by the threads. This mechanism is similar to the ragged synchronization of the Staccato concurrent copying GC [26].

```
1 handshake():
2   foreach t in all_threads:
3     if t != current_thread: t.handshake_request = TRUE
4   foreach t in all_threads:
5     if t != current_thread:
6       wait until t.handshake_request == FALSE
```

Figure 6. Handshake.

Figure 6 shows the pseudo code of the copier side of the handshake. The copier sends a handshake request to all other threads by setting the `handshake_request` flag in the thread structures on line 3. All threads periodically check the request and send an acknowledgment by toggling the flag. The handshake flag is checked at the GC safe points.

In the implementation, we used a handshake infrastructure of the HotSpot VM. Although the infrastructure is more sophisticated to deal with suspended threads, threads that execute native functions, and other waiting threads, it is the same as the pseudo code in essence.

3.4 Determining Transitive Closure

The mark phase determines the transitive closure of the closure root, `r`. In the rest of this section, we use the colors of the tri-color abstractions [12] to denote the states of *volatile* objects; unmarked objects are *white*, marked objects that may have references to white objects are *gray*, and the other marked objects are *black*. We do not use color for persistent objects.

`mark_transitive_closure` called on line 9 in Figure 7 is the central function in the mark phase. It traverses the object graph from the objects in the `worklist`. At the first call, `worklist` has only the closure root. It uses the `worklist` as a mark stack and terminates when the `worklist` becomes empty. Note that the replica is installed on the object by `shade` at line 22 (see Section 2.2.3 for details).

At line 10 the mark phase performs handshaking. From now on, the writers recognize all replicas installed by `mark_transitive_closure`.

3.5 Concurrent Mutation

During the traversal, all visited objects are accumulated in an array `gray_array`. We call this array the gray array because during the mark phase, all objects in this array are considered gray.

In a multi-threaded environment, another thread may install a reference to a white object into a marked object while `mark_transitive_closure` is traversing. The mark phase scans the gray array to find such white object by calling `find_unmarked` on line 11. Unmarked objects discovered are added to the `worklist`. If some objects are added to the gray array, the mark phase repeats; `mark_transitive_closure` traverses from the added objects. Note that, in our evaluation in Section 5, it was very rare and at most three times in a single run of a benchmark program that unmarked objects are found.

```
1  make_persistent(r):
2    local worklist
3    local gray_array
4    if shade(r) == TRUE:
5      worklist.add(r)
6      gray_array.add(r)
7      // mark phase
8      while not worklist.empty():
9        mark_transitive_closure(worklist, gray_array)
10       handshake()
11       find_unmarked(worklist, gray_array)
12     // copy phase
13     foreach o in gray_array: copy_to_replica(o)
14   SYNC()
15
16 // mark objects in worklist and their transitive closure
17 mark_transitive_closure(worklist, gray_array):
18   while not worklist.empty():
19     p = worklist.remove()
20     foreach f in p.fields:
21       o = p[f]
22       if shade(o) == TRUE:
23         worklist.add(o)
24         gray_array.add(o)
25
26 find_unmarked(worklist, gray_array):
27   foreach p in gray_array:
28     foreach f in p.fields:
29       o = p[f]
30       if o.replica == null:
31         // o is not marked
32         if shade(o) == TRUE:
33           worklist.add(o)
34           gray_array.add(o)
```

Figure 7. Copier-wait version of `make_persistent`; function `shade` is from Figure 3.

If no objects are added to the `worklist`, the mark phase ends. This is safe because, once the copier performs a handshake, no more references to white objects are written to the objects that are marked before the handshake. When a writer thread writes such a reference to an object O, it finds that O has a replica pointer on line 5 in Figure 2. Therefore, the writer makes the white object persistent before writing the reference to O.

It is also possible that a reference to a marked object is overwritten and the marked object becomes *stray object*. However, the copier can find those stray objects because of the gray array.

3.6 Correctness Argument

In RBP, all writes to the DRAM copy of an object must eventually be propagated to the NVM copy. We show that this also holds in the copier-wait approach, under the x86 TSO memory consistency model, where two reads or two writes made by the same thread are not reordered.

Under x86 TSO, in our proposed scheme, writes made before a handshake are guaranteed to be visible to other threads after the handshake. This is because during handshake, the writer and the copier, both write to the `handshake_request` flag, and ensure that they are visible to each other.

The copier performs the handshake between the installation of the replica pointer and copy of the fields. Therefore, in the case where the writer writes before answering the handshake, the write is visible to the copier when the copier copies. In the case where the writer writes after answering the handshake, the replica pointer is visible to the writer, and the writer propagates the write. In either case, the write is propagated to the NVM copy (by the copier or writer).

4 Reducing the Overheads of Write Barriers using Static Analysis

In this section, we present a static analysis based technique to reduce the overheads of write barriers by reducing the cost of the copier-wait write barriers discussed in Section 3. We first describe the motivation for using static analysis for scheme, and then present the details of our proposed static analysis components.

4.1 Motivation

In the copier-wait approach we discussed in Section 3, whenever a thread tries to make objects persistent in a `putfield` instruction, it performs handshake with all the remaining threads, because some of those threads may write to the object being made persistent and may miss the installed replica pointers. However, if all the objects we are trying to make persistent are thread-local or already persistent, then this handshake can be skipped. This is because, if the objects are thread-local, no other threads can write to them, and if the objects are already persistent, the `putfield` does not have to make them persistent again.

In this situation, we can further elide two other overheads in `make_persistent`: (i) we can simplify the procedure to copy the contents of an object from DRAM to NVM in `copy_to_replica` called on line 7 in Figure 3. (ii) We can also simplify the `shade` function in Figure 3 so that we do not manage the responsible thread and elide the `SYNC` operation on line 14 in Figure 7 because no other thread may be making the same object persistent simultaneously. We call these three overheads as the *persistence-related overheads*.

For example, in Figure 1, let us assume that all the objects reachable from parameters `k` and `v` are thread-local. The field writes for the Lines 6, 7 and 9 do not take the slow path in the write barrier as the object being written to is thread-local (not persistent). The `assoc` function writes to a persistent object at line 11, and tries to make objects reachable from its right-hand-side (rhs), `r`. However, we can avoid the persistence-related overheads here because the object pointed by `r` and all the objects reachable from its `head` field are thread-local, and the value in its `tail` field is the reference loaded from a persistent root `Program.assocList` at line 10, which points to a persistent object. Note that all the objects reachable from a persistent object are persistent by the invariant of orthogonal persistency.

However, in the same example, say the object pointed to by `p` is passed to a different thread (not shown) before line 9, then we cannot avoid the persistence-related overheads at lines 9, and 11. In general, our target is not just the object pointed to by the right-hand-side (rhs) of such a field-store instruction, but the list of all the objects that are reachable from the rhs. We call this list as the points-to-closure, which abstracts the transitive closure of the reachable objects at runtime.

Note that the traditional thread-escape analysis is insufficient to reduce the persistence-related overheads, because for us to do so, the points-to-closure of the rhs should contain only thread-local objects. For example, for the code shown in Figure 1, as discussed in Section 2.3, the traditional thread-escape analysis will infer that the static field `Program.assocList` will point to a thread-escaping object, and hence we cannot elide the overheads at Lines 10 (or 11).

4.2 Persistence-Aware Escape Analysis

In this section, we extend the idea of classical escape analysis, discussed in Section 2.3, to handle persistent objects. We call this analysis, persistence-aware escape analysis. Besides tracking the objects that may explicitly escape to other threads (via the reachable objects to the thread-objects), we also track which objects are reachable from the persistent roots via, one or more field dereferences; such objects are also referred to as persistent objects.

In addition to the abstract objects defined for the standard escape analysis, we use one additional special abstract object $P_{obj} \in spl$, that summarizes all the persistent objects. Static loads from the persistent roots returns P_{obj} as it is guaranteed that object was made persistent earlier when it became reachable via a persistent static field.

In addition to the sets and maps shown in Figure 4, at each program-point, we maintain a set named *Persistent* that contains the objects which may be reachable from the persistent roots. Note that the sets *Persistent* and *Escape* will never include the special objects E_{obj}, and P_{obj}, respectively.

In Figure 8, we give brief details of flow functions, where we differ from the standard rules shown in Figure 5. Further, note that we use the above modified `SplObj` set for the rules in Figure 5, as well.

In case of field writes (of the form $x.f = y$), in addition to the rules shown in Figure 5, if the object pointed to by x may be persistent, then this information is propagated to all the objects in the points-to-closure of y.

While the rules for handling of (non-persistent) static reads/writes do not differ from that shown in Figure 5, the persistent ones do. Reading a persistent static field always returns P_{obj}. Writing to a persistent static field, leads to the propagation of persistence information on the points-to-closure of the rhs. If the static field was not marked as a persistent field, we handle the operations in the standard way: the objects may thread-escape.

Statement	Rule
(store) $x.f = y$	3. if $\rho_{in}(x) \cap Persistent \neq \phi$ propagate(y,$Persistent$,P_{obj})
(persistent static rd) $x = A.g$	$\rho_{out}(x) = \{P_{obj}\}$
(persistent static wr) $A.g = x$	propagate(x, $Persistent$, P_{obj})

Figure 8. Flow functions for the persistence-aware escape analysis; to be used in addition to the rules in Figure 5.

Using the Analysis Results For a `putfield` instruction of the form $x.f = y$, if the points-to-closure of y contains only objects that are either thread-local, or persistent then we elide the instructions corresponding to the persistence-related overheads.

For example, for the code shown in Figure 1, the object returned by the persistent static read (`Program.assocList`) is P_{obj}, and the persistence-related overheads can be elided for Lines 10. Similarly, for Line 11, let us focus on the part of the abstract stack and heap reachable from `r`: $\rho = \{\texttt{r} \rightarrow \{O_8\}\}$, $\sigma = \{O_8.\texttt{head} \rightarrow O_5, O_8.\texttt{tail} \rightarrow P_{obj}, O_5.\texttt{head} \rightarrow O_k, O_5.\texttt{tail} \rightarrow O_v\}$. It can be seen that all of these objects are either thread-local or persistent, and hence the persistence-related overheads can be elided here as well.

4.3 Implementation

We now describe some details in our implementation to make our analysis sound when we cannot analyze certain parts of the code (for example, library calls), or when we cannot track the precise points-to graph (for example, arrays).

Modelling arrays Modelling objects stored in an array is challenging as statically it is difficult to precisely model the array index. To address this complexity, we associate a type-less abstract field `summaryF` with each array. This field may point to any object that may be stored in the array.

Handling Library methods Analyzing Java library classes during static analysis has one main issue: the available library during static analysis may not match that available during the program execution (as the program may be running on an entirely different system than that used by the static analyser). This difference in implementations restricts us from statically analyzing library methods precisely. Handling the library methods naively (all objects reachable from the arguments are assumed to escape) may lead to overly conservative results. To handle such a scenario, we came up with an approach that depends on the inputs of the programmer. For analyzing each benchmark, our tool takes two lists as input: list of *safe*, and *updateFree* library methods. We call a library method as *safe* if it does not change the escape status of objects passed as parameters from thread-local/persistent to escaping. Similarly, we call a library method as *updateFree*, if it does not update the heap of any object reachable from the parameters. If a library method is marked as *safe* then we can summarize the effect on all the objects reachable from the parameters, by adding a special object called S_{obj}($\in$ `AObjs`) to the points-to set of all the fields of these objects. This summarizing can be skipped for library methods which are marked as *updateFree*. If the method is not marked as *safe* or *updateFree*, we mark all objects reachable from parameters as escaping and add E_{obj} to the points-to set of all the fields of these objects. Finally the return value of *safe* library methods is set to S_{obj}; for all the remaining library methods, the return value is E_{obj}.

Special treatment of S_{obj}: Like we maintain a set for escaping/persistent objects at each program point, we also maintain a set of summarized objects. If S_{obj} is marked as escaping, we mark all the summarized objects as escaping. While processing the load and store statements (Figure 8), we handle S_{obj} and the Summarized object set like we handle E_{obj}/P_{obj} objects, and *Escape*/*Persistent* sets (rules not shown).

4.4 Persistence-Aware Escape Analysis for a JVM

Considering the prohibitively high costs of points-to and escape analyses none of real world JVM include these analyses (in an elaborate manner) as part of their JIT compilation; at most they use some rough heuristics to compute the same, which leads to highly imprecise results. To address this issue, we take inspiration from prior work [37], and propose to perform these analysis as part of the static compilation and use the analysis results in the JIT compiler to reduce the overheads of the write barriers, without impacting the cost of JIT compilation time.

To maintain the correspondence between the static analysis results and the JVM, we use the method and the byte-code-index (instead of source code line number) to distinguish the abstract objects allocated in each method.

5 Evaluation

We implemented the runtime components of our proposed scheme (copier-wait approach) in the OpenJDK HotSpot VM extended with replication based persistence (RBP). We implemented the static components of our proposed scheme in the Soot compiler framework [40]. We use Tamiflex [8] to handle reflection and make it usable by Soot.

We present an evaluation answering the following research questions:

- RQ1 Can the proposed copier-wait write barrier scheme improve the performance of programs that rarely make object persistent?
- RQ2 What is the overhead of the handshaking scheme for programs that frequently make object persistent?
- RQ3 What is the impact of static analysis towards the elision of reduction of the handshaking overheads?

5.1 Benchmark Programs

We developed three benchmark programs (XML, HAMT, and CHAMP) representing our typical use cases of the proposed system [4, 15, 34, 36].

XML represents programs that use a persistent container structure. It parses a protein database [27] in a semi-stream manner where it parses a subtree for a single protein to a DOM using the Java API for XML Processing (JAXP). For each subtree, it creates a blob of object containing some information extracted from the DOM, and stores the object in a persistent container. The DOMs and the blobs of objects are created in DRAM and only the blobs objects are made persistent when they are stored in the persistent container. Handshakes are not needed for making the blobs persistent, because the blobs consist only of thread-local objects before they are stored.

HAMT uses an immutable data structure, hash-array mapped tries (HAMT) [4], as a persistent container. It excessively adds key-value pairs to the HAMT. A new HAMT with added key-value pairs shares objects of the old version of HAMT, which are persistent. Therefore, the transitive closure of the rhs of the assignment that makes the HAMT persistent contains persistent objects as well as thread-local objects. Handshakes are still not needed. To assess the effect of volume of objects made persistent at once (more precisely, the volume of objects made persistent per call to `make_persistent` in Figure 7), we varied the number of key-value pairs added in batch; HAMT-*n* adds *n* key-value pairs in batch, which are made persistent at the same time. We ported the implementation[38] from the Functional Java project [18] CHAMP is similar to the HAMT benchmark, but it uses the compressed hash-array mapped prefix-trees (CHAMP) [34] instead of HAMT. We used the library in the Capsule Hash Trie Collections Library [35] for the implementation of CHAMP. For both HAMT and CHAMP, we use two values for *n* (1 and 10) for the evaluation.

We also used the DaCapo benchmarks [6] (version DaCapo-9.12-MR1-batch) to examine the performance of a wider range of applications. We used the default settings of the DaCapo except for the number of iterations noted below. Note that we excluded batik, eclipse, tomcat, tradebeans, and tradesoap because RBP [25] reported that they were unable to run them with the version of the standard HotSpot VM to which they implemented RBP.

Table 1 shows the statistics of the benchmark programs, showing the elapsed times for a single iteration of the benchmark programs, the number of calls to `make_persistent`, the volume of objects made persistent, and the average number of threads involved in a handshake; this is the number of threads to which a handshake request is sent. Note that all the numbers in this table are collected from the copier-wait approach implementation with no analysis and the all-durable setting.

5.2 Methodology

To assess the effectiveness of removing the MFENCE instruction from the hot path of the write barrier, we executed benchmarks in the non-durable setting where all the `@durable_root` annotations are ignored and no objects are made persistent. In this setting, the write barriers always take the hot path.

To assess the overhead of handshakes and the effectiveness of our analysis to reduce the persistence-related overheads, we also executed benchmarks in the all-durable setting. The all-durable setting considers all static variables as if they were annotated as persistent roots. We used this setting rather than following the `@durable_root` annotation because the DaCapo benchmarks are not annotated, and the only static variables are the root of the persistent container and a single `Object` for locking, whose effect is negligible.

For all benchmark programs, we executed 30 iterations in a single run and computed the average elapsed times for the last 25 iterations. For the number of events such as the number of handshakes, we counted them in the second iteration.

We configured OpenJDK 16 HotSpot VM to use the C1 compiler but not the C2 compiler. We also configured it to use the serial GC because RBP only supported the serial GC. We compiled it with GCC version 9.4.0 compiler and executed it in Ubuntu 20.04.4 LTS running on a computer with two Intel Xeon Gold 6354 processors (18 cores per socket) and 188 GB of DRAM. The NVM hardware we used is the Intel Optane DC persistent memory 200 series with 256 GB capacity per module. To avoid the noise due to the memory access latency caused by the interconnect in a NUMA environment, we used a single CPU by setting thread affinity and the NVM directly connected to the CPU. In our environment, a single NVM module was connected to a CPU.

5.3 RQ1: Effectiveness of Removing MFENCE

Figure 9 compares the elapsed times for the writer-wait approach (RBP) and copier-wait approach (our proposal) in the none-durable setting normalized to the standard HotSpot VM. For this part of evaluation, we did not elide operations causing persistence-related overheads. A lower number indicates a more efficient performance. The overhead for the writer-wait approach was 43.7% on average.

For all benchmarks, the proposed copier-wait approach was consistently better than the writer-wait approach, although the differences were within the 1× standard deviation for immutable data structures, HAMT and CHAMP, and jython. Performance improvements were 23.0% on average.

The remaining overhead for the copier-wait approach was 10.6% on average. There is no strong correlation between the overhead for the writer-wait and copier-wait approaches. Therefore, the cause of the remaining overhead was unlikely to be write barriers. Presumably, it came from the per object extra header word, which not only requires initialization at object allocation but also affects GC. Addressing this overhead is left as future work.

Table 1. Statistics of benchmark programs. Numbers are counted using the copier-wait approach implementation with no analysis and the all-durable setting.

program	time [sec]	`make_persistent`		volume of persistent objs			involved
		$\times 10^3$ call	$\times 10^3$ call/sec	$\times 10^6$ word	word/call	$\times 10^3$ word/sec	threads (avg.)
XML	21.3	525.3	24.6	170.2	324.1	7981.2	35.0
HAMT-1	19.9	1000.0	50.1	189.1	189.1	9481.3	7.0
HAMT-10	10.4	100.0	9.6	144.6	1446.1	13940.0	7.0
CHAMP-1	21.4	1000.0	46.7	202.2	202.2	9435.6	7.0
CHAMP-10	11.1	100.0	9.0	151.3	1513.3	13679.6	7.0
avrora	21.6	305.5	14.1	2.4	7.8	110.7	11.0
fop	0.5	0.0	0.0	0.0	72.2	2.6	5.0
h2	27.7	1101.3	39.8	35.4	32.1	1276.8	32.8
jython	52.3	4893.5	93.5	126.5	25.9	2418.9	6.0
luindex	10.9	24.7	2.3	2.3	93.9	213.7	5.0
lusearch	15.9	672.7	42.2	12.2	18.1	763.3	37.2
pmd	1.2	15.8	12.9	10.0	634.9	8191.2	37.5
sunflow	0.6	0.3	0.4	0.2	601.5	264.0	19.8
xalan	19.3	760.8	39.4	95.3	125.3	4942.7	40.7

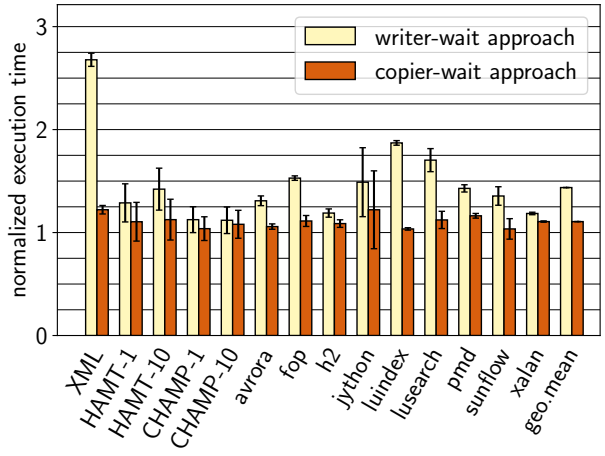

Figure 9. Elapsed times in none-durable setting normalized to the standard HotSpot VM. The error bars indicate the standard deviation.

5.4 RQ2: Handshake Overhead

Figure 10 shows the breakdown of the elapsed times for the copier-wait approach. The colored parts show the elapsed times for implementations where all the handshakes, including necessary ones, were omitted. Therefore, the remaining parts (white parts) show the handshake overhead. Note that the implementations without handshakes are not sound; although we observed that incidentally executions completed with correct answers.

The colored part can be below 1.0 in Figure 10 due to the benefit of removing the MFENCE instruction that we discussed in Section 5.3. Even with the `all-durable` settings, write barriers writing to fields of non-persistent objects take the hot path. Those that write primitive values also take the hot path. Therefore, the removal of the MFENCE instruction was effective in the case where the program makes objects persistent.

In the rest of this subsection, we focus on the left bars, where static analysis for reducing persistence-related overheads was not applied. We have observed that the overheads due to handshaking, formed the major part of the persistence-related overhead, and hence we give additional focus to the number of handshakes performed. In this evaluation, all calls to `make_persistent` performed `handshake` exactly once; no unmarked objects were found after handshake on line 11 in Figure 7. Therefore, the frequencies of the handshakes were equal to the numbers of calls to `make_persistent` per second shown in the fourth column of Table 1.

The handshake overhead was 37.9% on average; the average was 29.4% if we exclude fop, h2, and jython. Benchmarks with high handshake frequencies showed large overheads. For example, jython had the highest handshake frequency and showed the highest overhead. Benchmarks with more than 50% of handshake overhead (HAMT-1, CHAMP-1, h2, jython, xalan) performed handshakes at 39.8×10^3 times per second or more.

However, lusearch did not show high handshakes overhead while performing handshakes at 42.2×10^3 times per second. We think this is because the performance bottleneck for lusearch was the bandwidth of the NVM module; handshakes were completed during previous writes to persistent objects (possibly by other threads) were being written to NVM. Matsumoto [25] reported that lusearch wrote to persistent objects excessively, at 25.7×10^6 writes per second in their experimentation. Izraelevitz [17] reported that the bandwidth for a single NVM device was around 2 GB/s at the

access size of 256 bytes or larger for random writes, and the NVM's access granularity was 256 bytes. This implies that a single NVM device is capable of processing up to 8×10^6 writes per second. Although Izraelevitz used the Optane Persistent Memory 100 series while we used 200 series, the bandwidth was likely to be saturated.

5.5 RQ3: Effectiveness of persistence-related Overheads Reduction

Figure 10 also shows the impact of static analysis. The right bars in the figure show the breakdown of elapsed times for the copier-wait approach with the compiler analysis to reduce persistence-related overheads. For fop, h2, and jython, we could not complete static analysis in 14 hours and hence skipped.

Figure 11 shows the number of handshakes. To improve the visibility all the numbers are divided by the elapsed time for the run without handshake elision. Note that the number of handshakes for the run with static analysis are also divided by the elapsed time for the run without it.

The static analysis removed 52.0% of the overheads and improved overall performance by 18.5% on average. This improvement includes the benefit of eliding synchronizations. As a result, the copier-wait approach with static analysis was almost as fast as the writer-wait approach; the copier-wait approach was slower by 2.4% on average.

The static analysis was effective to the benchmarks representing our targets: XML representing programs with persistent containers and HAMT and CHAMP representing programs with immutable data structures. For HAMT and CHAMP, it elided almost all handshakes, and as a result, all overheads were eliminated. For XML, half of the handshakes were elided. Because XML used the JAXP library, we could not analyze precisely. The conservative analysis left the remaining handshakes.

For HAMT and CHAMP, the run with static analysis was even faster than the run where all handshakes are omitted, including the necessary ones (colored part in Figure 10). This shows the effectiveness of the elision of synchronizations in `make_persistent`.

The handshake elision was effective for some of the DaCapo benchmarks as well. lusearch showed an exceptional behavior; although parts of handshakes were elided, the run with static analysis was faster than the run where all handshakes are omitted, including necessary ones (colored part in Figure 10). We think this is because of the reduction of writes to NVM performed for synchronization in `make_persistent` under the situation where the NVM bandwidth was saturated. `make_persistent` writes the thread ID of responsible thread to replicas in NVM on line 12 in Figure 3 and clears it at the end in `SYNC`. The `copy_to_replica` function also reads replica.

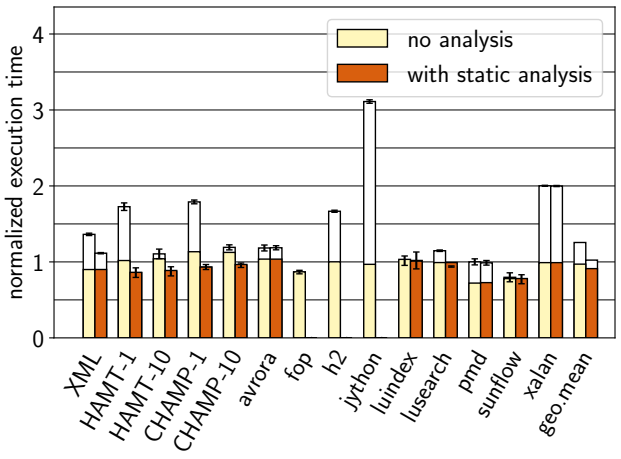

Figure 10. Elapsed times for the copier-wait approach in all-durable setting normalized to the writer-wait approach. White parts indicate the elapsed time for handshakes. Results for the rightward bars for fop, h2, and jython are unavailable. These benchmarks are excluded from geometric mean for both bars. The error bars indicate the standard deviation.

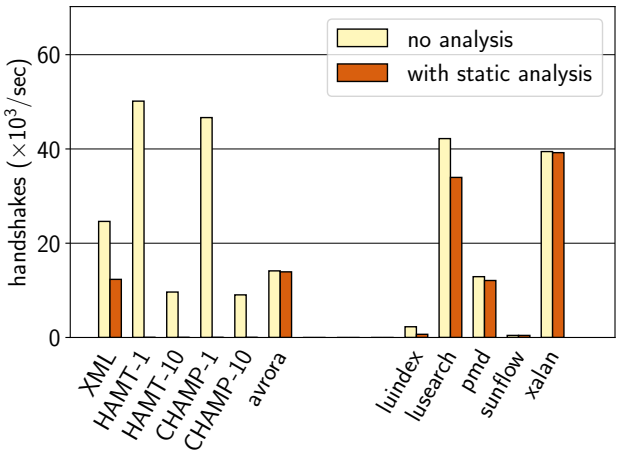

Figure 11. The number of handshakes; to improve the visibility all the numbers are divided by the elapsed time for the respective run without static analysis. Benchmarks for which handshake reduction was not available are not shown.

5.6 Threats to Validity

We briefly discuss two main threats to validity and how we mitigate them.

- The idea of the proposed copier-wait approach is based on an intuition that the number of persistent writes are typically not too many and thus using an expensive handshake in the cold-path will not overley impact the performance negatively. And further, among the persistent writes, we can use static analysis to eliminate the write-barriers (handshake) in many cases. One can argue that we can write programs where most

of the writes happen to persistent memory and the static analysis may not be able to elide the handshakes. While this is true in general, in practice we have found that writes to persistent writes (e.g., objects reachable via static fields) are not too frequent. Further, we argue that such writes are typically grouped together and our proposed static analysis can be used to elide the handshakes.

- XML, HAMT and CHAMP are manually written codes and may not represent the regular programs. Mitigation: These benchmarks are deliberately developed to evaluate certain types of behaviors. Thus their not being general purpose programs is not a major concern, as we do use other general purpose benchmarks (popular benchmarks from DaCapo) to give a broader perspective.

6 Related Work

The orthogonal persistence model was proposed about 30 years ago [3] and was explored further by many researchers[1, 2, 13, 24]. Recently two Java VMs implementing orthogonal persistence for NVMs were proposed, which extended the work of Shull et al. [30]: AutoPersiste [31] and RBP [25]. They use NVM for storage and make updates to objects persistent immediately, without programmers specification. Furthermore, they make objects persistent on-the-fly, that is, without stopping the world.

Although these properties are attractive, they come with a large overhead applied to write barriers. QuickCheck [32] addressed this problem by moving the slow path of the write barrier away from the hot path to improve code locality. P-INSPECT Kokolis et al. [20] used a bloom filter implemented in a custom processor to handle checks in the write barrier in hardware. However, both did not address the MFENCE needed on the hot path in write barriers that we eliminate.

In our new write barrier, we use handshakes. In the context of concurrent copying GC, similar techniques are used. Staccato GC [26] performs handshakes to ensure that the copying marks become visible to mutators in Power PC processors, which have a weaker memory consistency model than the x86 TSO. CHICKEN [29] used the handshake for the same purpose in x86 computers. The handshake is also used to eliminate an atomic instruction from the hot path of the locking bytecode [19, 41].

Denny et al. [11] present a language extension called NVL-C which can be used by the programmer to specify different directives so that consistency of the persistent storage can be maintained. In contrast, our compiler analysis can be used to reduce the persistence-related overheads. We believe that our technique can be (suitably extended and) used along with NVL-C, to reduce the overheads in their programs.

There have been prior works that analyze and verify persistent memory programs for different properties. For example, Bansal [5] presents a static analysis tool to check that in the user written program the persistent storage has no pointers to the non-persistent storage. Dahiya and Bansal [10] present a tool to verify the correctness of an intermittent program, with respect to its continuous counterpart. In contrast to these works, we present a novel analysis to reduce the overheads of the write barriers.

There have been many prior works [7, 9, 21, 33, 37, 42] that use points-to and escape analysis to optimize parallel Java programs, by eliminating redundant synchronization operation, stack allocation of objects, and scalar replacement. Points-to analysis and escape analysis forms an important basis for computing MHP analysis [28] which is used to reason about data-races and deadlocks in parallel programs. In this paper, we present a novel extension to escape analysis, called persistence-aware escape analysis that can be used to reduce the overheads of the write barriers.

7 Conclusion and Future Work

In this paper, we proposed two approaches to reduce the synchronization overhead in the write barriers of modern implementations of the orthogonal persistence model. The first approach proposes a runtime technique (in terms of a new write barrier), where we removed the expensive MFENCE from the hot path of the write barrier. Although this new write barrier reduced overhead for programs that rarely make objects persistent, it introduced a new large overhead of handshake to programs that frequently make objects persistent. The second approach proposes an extension to escape analysis called persistence-aware escape analysis to reduce the handshake and related overheads. Our evaluation shows that when the programmer does not specify persistent roots, our runtime technique is able to improve the performance by an average 23.0%. We also show that when the programmer specifies all the static variables as persistent roots, our static analysis aided scheme leads to significant improvements (18.5% on average). The actual impact depended on the specifics of the benchmarks under consideration.

We believe that the proposed copier-wait approach can also be used in the context of AutoPersist [31]. Further, the proposed persistence-aware escape analysis can also be applied there, to elide the corresponding redundant handshakes, along with possible read barriers and equality testing instructions. Considering the large engineering efforts involved in this whole scheme, we leave it as a future work.

Acknowledgments

The authors would like to thank the Sakura Science Program for helping the setting up of this collaboration. Parts of this work are supported by JSPS through JSPS KAKENHI three grant numbers: JP22H03566, JP23K24822, and JP24H00688. The second and third authors would like to acknowledge IBM CAS funding (Award number 1156) for partially supporting this research.

References

[1] Malcolm P. Atkinson and Mick J. Jordan. 1999. Issues Raised by Three Years of Developing PJama: An Orthogonally Persistent Platform for Java. In *Proceedings of the 7th International Conference on Database Theory (ICDT 1999) (Lecture Notes in Computer Science, Vol. 1540)*. Springer, 1–30. https://doi.org/10.1007/3-540-49257-7_1

[2] Malcolm P. Atkinson, Mick J. Jordan, Laurent Daynès, and Susan Spence. 1996. Design Issues for Persistent Java: A Type-Safe, Object-Oriented, Orthogonally Persistent System. In *Proceedings of the 7th Workshop on Persistent Object Systems*. Morgan Kaufmann, 33–47.

[3] Malcolm P. Atkinson and Ronald Morrison. 1995. Orthogonally Persistent Object Systems. *VLDB J.* 4, 3 (1995), 319–401. https://doi.org/10.1007/BF01231642

[4] Phil Bagwell. 2001. *Ideal Hash Trees*. Technical Report.

[5] Sorav Bansal. 2023. StaticPersist: Compiler Support for PMEM Programming. In *Verification, Model Checking, and Abstract Interpretation: 24th International Conference, VMCAI 2023, Boston, MA, USA, January 16–17, 2023, Proceedings* (Boston, MA, USA). Springer-Verlag, Berlin, Heidelberg, 44–65. https://doi.org/10.1007/978-3-031-24950-1_3

[6] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[7] Bruno Blanchet. 2003. Escape analysis for JavaTM: Theory and practice. *ACM Trans. Program. Lang. Syst.* 25, 6 (nov 2003), 713–775. https://doi.org/10.1145/945885.945886

[8] Eric Bodden, Andreas Sewe, Jan Sinschek, Mira Mezini, and Hela Oueslati. 2011. Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. In *Proceeding of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. ACM, New York, NY, USA, 241–250. https://doi.org/10.1145/1985793.1985827

[9] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. 1999. Escape Analysis for Java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Denver, Colorado, USA) *(OOPSLA '99)*. Association for Computing Machinery, New York, NY, USA, 1–19. https://doi.org/10.1145/320384.320386

[10] Manjeet Dahiya and Sorav Bansal. 2018. Automatic Verification of Intermittent Systems. In *Verification, Model Checking, and Abstract Interpretation*, Isil Dillig and Jens Palsberg (Eds.). Springer International Publishing, Cham, 161–182.

[11] Joel E. Denny, Seyong Lee, and Jeffrey S. Vetter. 2016. NVL-C: Static Analysis Techniques for Efficient, Correct Programming of Non-Volatile Main Memory Systems. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing* (Kyoto, Japan) *(HPDC '16)*. Association for Computing Machinery, New York, NY, USA, 125–136. https://doi.org/10.1145/2907294.2907303

[12] Edsger W. Dijkstra, Leslie Lamport, Alain J. Martin, Carel S. Scholten, and Elisabeth F. M. Steffens. 1978. On-the-Fly Garbage Collection: An Exercise in Cooperation. *Commun. ACM* 21, 11 (1978), 966–975. https://doi.org/10.1145/359642.359655

[13] Jorge Guerra, Leonardo Mármol, Daniel Campello, Carlos Crespo, Raju Rangaswami, and Jinpeng Wei. 2012. Software Persistent Memory. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, Gernot Heiser and Wilson C. Hsieh (Eds.). USENIX Association, 319–331. https://www.usenix.org/conference/atc12/technical-sessions/presentation/guerra

[14] Shashin Halalingaiah, Vijay Sundaresan, Daryl Maier, and V. Krishna Nandivada. 2024. The ART of Sharing Points-to Analysis: Reusing Points-to Analysis Results Safely and Efficiently. In *Proceedings of the 2024 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2024, part of SPLASH 2024, Pasadena, CA, USA, October 2024 (OOPSLA '24)*. Association for Computing Machinery, New York, NY, USA.

[15] Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2020. MOD: Minimally Ordered Durable Datastructures for Persistent Memory. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 775–788. https://doi.org/10.1145/3373376.3378472

[16] Intel. 2023. *Intel® 64 and IA-32 Architectures Software Developer's Manual*.

[17] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amir Saman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR* abs/1903.05714 (2019). arXiv:1903.05714 http://arxiv.org/abs/1903.05714

[18] Functional Java. 2022. Functional programming library for Java. https://github.com/functionaljava.

[19] Kiyokuni Kawachiya, Akira Koseki, and Tamiya Onodera. 2002. Lock reservation: Java locks can mostly do without atomic operations. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002, Seattle, Washington, USA, November 4-8, 2002*, Mamdouh Ibrahim and Satoshi Matsuoka (Eds.). ACM, 130–141. https://doi.org/10.1145/582419.582433

[20] Apostolos Kokolis, Thomas Shull, Jian Huang, and Josep Torrellas. 2020. P-INSPECT: Architectural Support for Programmable Non-Volatile Memory Frameworks. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020*. IEEE, 509–524. https://doi.org/10.1109/MICRO50266.2020.00050

[21] Thomas Kotzmann and Hanspeter Mossenbock. 2007. Run-Time Support for Optimizations Based on Escape Analysis. In *International Symposium on Code Generation and Optimization (CGO'07)*. 49–60. https://doi.org/10.1109/CGO.2007.34

[22] Yi Lin, Kunshan Wang, Stephen M. Blackburn, Antony L. Hosking, and Michael Norrish. 2015. Stop and go: understanding yieldpoint behavior. In *Proceedings of the 2015 ACM SIGPLAN International Symposium on Memory Management, ISMM 2015, Portland, OR, USA, June 13-14, 2015*, Antony L. Hosking and Michael D. Bond (Eds.). ACM, 70–80. https://doi.org/10.1145/2754169.2754187

[23] Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005)*. ACM, 378–391. https://doi.org/10.1145/1040305.1040336

[24] Leonardo Mármol, Mohammad Chowdhury, and Raju Rangaswami. 2018. LibPM: Simplifying Application Usage of Persistent Memory. *ACM Trans. Storage* 14, 4 (2018), 34:1–34:18. https://doi.org/10.1145/3278141

[25] Kotaro Matsumoto, Tomoharu Ugawa, and Hideya Iwasaki. 2022. Replication-based object persistence by reachability. In *ISMM '22: ACM SIGPLAN International Symposium on Memory Management, San Diego, CA, USA, 14 June 2022*, Michael Lippautz and David Chisnall (Eds.). ACM, 43–56. https://doi.org/10.1145/3520263.3534653

[26] Bill McCloskey, David F. Bacon, Perry Cheng, and David Grove. 2008. *Staccato: A Parallel and Concurrent Real-time Compacting Garbage Collector for Multiprocessors*. IBM Research Report RC24505. IBM Research. http://domino.watson.ibm.com/comm/research_people.

nsf/pages/dgrove.rc24504.html

[27] Gerome Miklau. 2001. Protein Sequence Database, PSD7003.xml. http://aiweb.cs.washington.edu/research/projects/xmltk/xmldata/www/repository.html.

[28] Gleb Naumovich and George S. Avrunin. 1998. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Lake Buena Vista, Florida, USA) *(SIGSOFT '98/FSE-6)*. Association for Computing Machinery, New York, NY, USA, 24–34. https://doi.org/10.1145/288195.288213

[29] Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. 2008. A Study of Concurrent Real-Time Garbage Collectors. In *29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. 33–44. https://doi.org/10.1145/1379022.1375587

[30] Thomas Shull, Jian Huang, and Josep Torrellas. 2018. Defining a high-level programming model for emerging NVRAM technologies. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes (ManLang 2018)*. ACM, 11:1–11:7. https://doi.org/10.1145/3237009.3237027

[31] Thomas Shull, Jian Huang, and Josep Torrellas. 2019. AutoPersist: an easy-to-use Java NVM framework based on reachability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, 316–332. https://doi.org/10.1145/3314221.3314608

[32] Thomas Shull, Jian Huang, and Josep Torrellas. 2019. QuickCheck: using speculation to reduce the overhead of checks in NVM frameworks. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2019, Providence, RI, USA, April 14, 2019*, Jennifer B. Sartor, Mayur Naik, and Christopher J. Rossbach (Eds.). ACM, 137–151. https://doi.org/10.1145/3313808.3313822

[33] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Partial Escape Analysis and Scalar Replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Orlando, FL, USA) *(CGO '14)*. Association for Computing Machinery, New York, NY, USA, 165–174. https://doi.org/10.1145/2581122.2544157

[34] Michael J. Steindorfer and Jurgen J. Vinju. 2015. Optimizing hash-array mapped tries for fast and lean immutable JVM collections. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 783–800. https://doi.org/10.1145/2814270.2814312

[35] Michael J. Steindorfer and Jurgen J. Vinju. 2023. The Capsule Hash Trie Collections Library. https://github.com/usethesource/capsule.

[36] Nicolas Stucki, Tiark Rompf, Vlad Ureche, and Phil Bagwell. 2015. RRB vector: a practical general purpose immutable sequence. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, Kathleen Fisher and John H. Reppy (Eds.). ACM, 342–354. https://doi.org/10.1145/2784731.2784739

[37] Manas Thakur and V. Krishna Nandivada. 2019. PYE: A Framework for Precise-Yet-Efficient Just-In-Time Analyses for Java Programs. *ACM Trans. Program. Lang. Syst.* 41, 3, Article 16 (jul 2019), 37 pages. https://doi.org/10.1145/3337794

[38] Tomoharu Ugawa. 2024. *Benchmarks*. https://github.com/plasgroup/ccjava-benchmarks

[39] Tomoharu Ugawa, Carl G. Ritson, and Richard E. Jones. 2018. Transactional Sapphire: Lessons in High-Performance, On-the-fly Garbage Collection. *ACM Trans. Program. Lang. Syst.* 40, 4 (2018), 15:1–15:56. https://doi.org/10.1145/3226225

[40] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: a Java bytecode optimization framework. In *CASCON First Decade High Impact Papers* (Toronto, Ontario, Canada) *(CASCON '10)*. IBM Corp., USA, 214–224. https://doi.org/10.1145/1925805.1925818

[41] Ting Wang, Michihiro Horie, Kazunori Ogata, Hao Chen Gui, Xiao Ping Guo, and Yang Liu. 2020. Enhancement of OpenJDK biased locking for infrequent lock contention. In *Programming'20: 4th International Conference on the Art, Science, and Engineering of Programming, Porto, Portugal, March 23-26, 2020*, Ademar Aguiar, Shigeru Chiba, and Elisa Gonzalez Boix (Eds.). ACM, 23–26. https://doi.org/10.1145/3397537.3397562

[42] John Whaley and Martin Rinard. 1999. Compositional Pointer and Escape Analysis for Java Programs. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Denver, Colorado, USA) *(OOPSLA '99)*. Association for Computing Machinery, New York, NY, USA, 187–206. https://doi.org/10.1145/320384.320400

Received 2024-07-01; accepted 2024-08-30